This paper appeared in: Frontiers'92: The Fourth Symposium on the Frontiers of Massively Parallel Computation, pages 178–185, McLean, Virginia, October 19–21, 1992

Automatic Data Distribution for Nearest Neighbor Networks

Michael Philippsen
Universität Karlsruhe
D-7500 Karlsruhe, F.R.G.
phlipp@ira.uka.de

Abstract

An algorithm for mapping an arbitrary, multidimensional array onto an arbitrarily shaped multidimensional nearest neighbor network of a distributed memory machine is presented. The individual dimensions of the array are labeled with high-level usage descriptors that can either be provided by the programmer or can be derived by sophisticated static compiler analysis.

The presented algorithm achieves an appropriate exploitation of nearest neighbor communication and allows for efficient address calculations.

We describe the integration of this technique into an optimizing compiler for Modula-2* and derive extensions that render efficient translation of nested parallelism possible and provide some support for thread scheduling.

1 Introduction

An important problem facing numerous projects on distributed memory machines is that of distributing array data over the available processors. There is widespread agreement about the two goals of data distribution: (1) Data locality. All data elements should be stored local to a processor that executes some statements upon it, reducing the amount of communication and resulting in minimal run-time. (2) Parallelism. Perfect data locality and the minimal communication cost are achieved by employing only a single processor. In general, however, the run-time can be improved by exploiting the full degree of parallelism provided by the hardware. An appropriate trade-off between the conflicting goals of data locality and parallelism must be found.

Whereas the goals are agreed upon, totally different approaches to reach them have been developed. In many programming languages the user must program the data layout explicitly. Some languages require an explicit mapping of the data onto the topology [19, 20, 12, 16, 2], others are more abstract and provide either sets of directives for the compiler or interactive or knowledge-based environments that help determine alignment of array dimensions and mapping functions [4, 13, 10, 18, 1, 14, 3]. Even a data distribution language has been developed for this purpose [7]. Recent work [8, 9, 5, 17, 11] focuses on static compile-time analysis to automatically find data decomposition that achieves both goals.

The approaches based on directives and on automatic decomposition often use a high-level intermediate representation where transformations are done: dimensions of different arrays are grouped together, super-arrays and corresponding index transfer functions are derived, usage patterns are exploited, etc. After this complex compiler phase the resulting data structures which can again be understood as arrays, are straightforwardly mapped onto the available processors of the hardware.

However, we feel that the mapping of the (resulting) arrays onto the topology could be improved to reach the following goals: (3) Exploit fast communication patterns if there is special hardware support for a particular communication pattern. (4) Simple address calculations. The computation of processor numbers and addresses of data elements must be fast. Otherwise, evaluation of complex expressions easily consumes too much computational power.

Therefore, we present an algorithm for mapping an arbitrary, multidimensional array onto an arbitrarily shaped multidimensional nearest neighbor network of a distributed memory machine.

After a short introduction of nearest neighbor networks and the notation we use, we identify some allocators that are commonly used. In the fourth section we present our strategy for determining the data distribution. The strategy consists of several steps each of which is based on the available information and refines the distribution function. In the last two sections, we describe the integration of this technique into

an optimizing compiler for Modula-2*, and derive extensions that render efficient translation of nested parallelism possible and provide some support for thread scheduling.

2 $\bar{k}$-Coordinate n-dimensional Nearest Neighbor Networks

In a $\bar{k}$-coordinate n-dimensional neighbor network $\bar{k}$-NN_n, each processor is connected to two neighbors in each of the n dimensions. In other words, each node participates in n different NN_1 subnetworks, one per dimension. The number of processors in each of these dimensions is given by $\bar{k} = (k_1, k_2, \ldots, k_n)$. In a NN_n the degree of each node is $2n$.

Some NN_n have been used successfully in parallel machines and, thus, have received special names. For example, a $\bar{k}$-NN_n with $\bar{k} = (2, 2, \ldots, 2)$ is called a hypercube. A two- or three-dimensional processor grid is a NN_2 or a NN_3, resp. For our purposes, it does not matter whether a given $\bar{k}$-NN_n is a hypertorus, where the NN_1 subnetworks are rings, or a hypermesh, where they are linear arrays. We restrict our considerations to $\bar{k}$-NN_n with dimensions that are powers of two, i.e., there exist κ_i such that $k_i = 2^{\kappa_i}$ for all $1 \leq i \leq n$.

The number of bits that are necessary to encode the address of a node along dimension i is given by κ_i. Along a dimension the nodes are numbered continuously. The complete address of a node in the network needs $\sum_{i=1}^{n} \kappa_i$ bits. It is constructed by concatenating the address segments of length κ_i. Bits are numbered as follows

$$\underbrace{0, 1, \ldots, \kappa_1 - 1}_{\kappa_1}, \underbrace{\kappa_1, \ldots, \kappa_1 + \kappa_2 - 1}_{\kappa_2}, \ldots, \underbrace{\sum_{i=1}^{n-1} \kappa_i, \ldots, \sum_{i=1}^{n} \kappa_i - 1}_{\kappa_n}$$

From the address of a given node the address of a neighboring node along dimension i can be derived by incrementing or decrementing the according segment of κ_i bits $(\kappa_{i-1}, \kappa_{i-1} + 1, \ldots, \kappa_i - 1)$, i.e., by adding $\pm 1 \cdot 2^{\kappa_{i-1}}$. We define $\kappa_0 = 0$ for notational convenience.

3 Array Allocation

By studying several approaches for explicit declaration of layout patterns [19, 20, 12, 16, 22, 1, 14, 4, 13, 18, 10, 21] and for intermediate representation of allocation information in modern compilers [8, 9, 5, 17, 11, 15], we realized that a set of three machine-independent types of distribution information is used most often.

Consider a m-dimensional array. Each of the dimensions can be labeled with either `spread`, `cycle`, or `local`. These hints for data distribution are either derived from the explicit specification by the user or they are a result of compiler analysis of reference patterns. The semantics of these hints are as follows:

- `spread`: Dimensions that are attributed `spread` are divided into segments, one for each of the available processors. A vector with l elements is assigned to P processors by allocating a segment of length $\lceil l/P \rceil$ to each processor. While utilizing all available processors, it minimizes the cost of nearest-neighbor communication.

- `cycle`: Dimensions labeled with `cycle` are distributed in a round-robin fashion over the available processors. In contrast to `spread`, `cycle` maximizes the cost of nearest-neighbor communication: neighboring array elements are always in different processors, leading to better processor utilization if a parallel algorithm operates on subsegments of a vector at a time.

- If either `spread` or `cycle` apply to several dimensions, the distribution of the data on the processors must try to generalize the above communication behavior and requirements for potential parallelism for all these dimensions.

- `local`: Array elements whose indices differ only in a dimension that is labeled `local` are associated with the same processor. This facility is used to *avoid* distribution of data in a given dimension. By "unrolling" all given `local` dimensions into one pseudo-dimension with a length that is the product of the lengths of the individual dimensions only one single `local` dimension must be considered.

We focus our considerations on an array with m dimensions. The vector $\bar{l} = (l_1, l_2, \ldots, l_m)$ denotes the number of elements in each of the dimensions. We assume that the dimensions are rounded up to the next power of two, i.e., there exist λ_j such that $l_j = 2^{\lambda_j}$ for all $1 \leq j \leq m$.

The number of bits that are necessary to encode the subscript of one dimension j is given by λ_j. In total, $\sum_{j=1}^{m} \lambda_j$ bits are required to encode all subscripts. Without loss of generality we consider the dimensions of the array to be sorted such that the following label-

ing holds:

$$j = \begin{cases} 1 \ldots s & : \quad \texttt{spread} \\ s+1 \ldots c & : \quad \texttt{cycle} \\ c+1 \ldots m & : \quad \texttt{local} \end{cases}$$

Note that replication and other distribution patterns, e.g. `wrapped(k)`, are reducible to the allocators given above, by adding additional address bits or by simply splitting the original array address in two segments.

4 Data Distribution

To distribute array data we present a mapping function that operates in four major steps. Mapping is based on bit representations of the addresses of data elements in a virtual address space. First, we decide how many bits of this virtual address space are mapped onto both the processor address and the local address. Second, we allot the limited number of bits in the processor address to dimensions of a given array. Third, we fulfill the intention of `spread` and `cycle` by deciding which of the address bits of the dimensions will go into the processor address. Finally, the bits in the processor address are reshuffled to align `spread` dimensions with axes in the network to support efficient nearest neighbor communication.

1. Processor Bits and Local Bits

The $\sum_{j=1}^{m} \lambda_j$ address bits that are necessary to encode the offset of the array elements must be mapped onto a real topology. Therefore, the virtual address is split into two parts on distributed memory machines:

- the number of bits used for the processor address:

$$b_p = \min(\sum_{i=1}^{n} \kappa_i, \quad \sum_{j=1}^{c} \lambda_j)$$

- the number of bits used for the local part of the address, referring memory cells in the local memory of the processors:

$$b_l = \begin{cases} \sum\limits_{j=c+1}^{m} \lambda_j & \text{if} \quad \sum\limits_{j=1}^{c} \lambda_j \leq \sum\limits_{i=1}^{n} \kappa_i \\ \sum\limits_{j=1}^{m} \lambda_j \quad - \quad \sum\limits_{i=1}^{n} \kappa_i & \text{otherwise} \end{cases}$$

In this paper we deal with the general case, where there are fewer processor bits available than are required in total by all `spread` and `cycle` dimensions. Therefore, the processor fraction of the address of a data element consists of $b_p = \sum_{i=1}^{n} \kappa_i$ bits and the local fraction consists of the remaining $b_l = \sum_{j=1}^{m} \lambda_j - b_p$ bits.

2. Allotting of Processor Bits

In step 1, we have determined how many of the bits that are required to address all array elements are implemented by the node number and how many are mapped to addresses in the local memory of the nodes. In step 2, we allot the available number of processor bits to individual dimensions of the array, i.e., given the number of bits that are necessary to encode a subscript in a particular dimension, we decide now how many thereof are mapped onto the processor address.

For each dimension j we find numbers $p_j \in I\!N$ and $q_j \in I\!N$ with $\lambda_j = p_j + q_j$. Let p_j denote the number of bits that are mapped onto bits of the processor address. The number of local bits is denoted by q_j.

We propose a distribution that reflects the relation of the number of elements in individual dimensions, i.e., a dimension that has twice the number of elements than another dimension, should receive a twice as many processor bits. One can think of other allotting schemes, e.g., a simple even distribution solely based on the number of dimensions or an allotment driven by weights that are based on detailed program analysis. For simplicity, however, we favor the following distribution:

$$p_j = \begin{cases} \min(\lambda_j, \quad \frac{l_j}{\sum\limits_{\iota=1}^{c} l_\iota} \cdot b_p) & \text{if} \quad 1 \leq j \leq c \\ \\ 0 & \text{if} \quad c+1 \leq j \leq m \end{cases}$$

The number of local bits q_j for a dimension $1 \leq j \leq m$ is thus $q_j = \lambda_j - p_j$. Work on comparing different allotting schemes and on deriving allotting information from programs is in progress.

3. Semantics of `spread` and `cycle`

Although for the address in each dimension the *number* of bits that end up in the processor address or in the local fraction of the address have been determined, it is still unclear to which group each particular bit belongs if both p_j and q_j are non-zero.

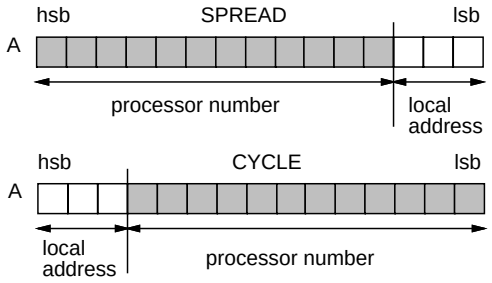
The selection is based on the semantics of `spread` and `cycle`:

- `spread`: by assigning the *first* p_j bits of the coordinate address to the processor bits, the semantics

of `spread` is fulfilled: neighboring data elements whose addresses differ only in the lowest bits reside in the same processor.

- `cycle`: for cyclic dimensions, the processor bits are assigned from the *rear*. Thus, neighboring data elements end up in different processors.

For example consider the data distribution for an array with one dimension (length $l_1 = 32,768$) on a machine with 4k processors, i.e., a 4k-NN_1. Although $\lambda_1 = 15$ bits are required to represent the address of an array element, only $\kappa_1 = 12 = p_1$ bits of processor number are available.



The figure shows the selection of those p_1 bits of the address that end up in the processor address (shaded) in case of both a `spread` and a `cycle` labeling of the dimension.

Assume a subscript vector $(x_1, x_2, \ldots, x_m)$. For the local address, each dimension with $q_j \neq 0$ contributes q_j bits. These bits are concatenated, i.e., shifted according to the sum of all q_ι of earlier dimensions and added together. For `spread` dimensions, the *last* q_j bits of x_j are extracted by computing $x_j \bmod 2^{q_j}$. The *first* q_j bits of a subscript x_j in a `cycle` dimensions are derived by $x_j \operatorname{div} 2^{p_j}$:

$$\text{localAdr} = \sum_{j=1}^{m} x'_j \cdot 2^{\sum_{\iota=1}^{j-1} q_\iota}$$

$$\text{with} \quad x'_j = \begin{cases} x_j \bmod 2^{q_j} & \text{if} \quad j \in \{1 \ldots s\} \\ x_j \operatorname{div} 2^{p_j} & \text{if} \quad j \in \{s+1 \ldots c\} \\ x_j & \text{if} \quad j \in \{c+1 \ldots m\} \end{cases}$$

Analogously, the preliminary processor address is derived. Here the div and mod operations are exchanged since for `spread` dimensions the first (div 2^{q_j}) and for `cycle` dimensions the last (mod 2^{p_j}) bits must be extracted.

$$\text{prelimProcessorNo} = \sum_{j=1}^{m} x''_j \cdot 2^{\sum_{\iota=1}^{j-1} p_\iota}$$

$$\text{with} \quad x''_j = \begin{cases} x_j \operatorname{div} 2^{q_j} & \text{if} \quad j \in \{1 \ldots s\} \\ x_j \bmod 2^{p_j} & \text{if} \quad j \in \{s+1 \ldots c\} \\ x_j & \text{if} \quad j \in \{c+1 \ldots m\} \end{cases}$$

Note that these address computations can be done by masking and shifting, i.e., by bit operations that can efficiently be executed on virtually all computers.

4. Network Topology

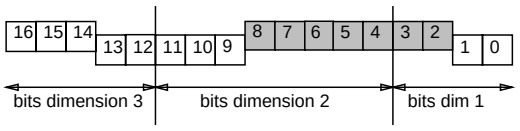
Although a preliminary processor address has been determined in step 3, this mapping does not use the topology of the network. Due to the semantics of `spread`, it is desirable to exploit as many of the n nearest neighbor links provided by a NN_n as possible for mapping of `spread` dimensions. This can be achieved by mapping the p_j bits of a `spread` dimension onto a segment of the processor address that represents a NN_1 subnetwork. Therefore, the p_j bits ideally should start at bit position 0, κ_1, ..., or κ_{n-1}.

Step 4 gives a permutation for the bits in the preliminary processor address. This permutation results in an ordering such that the least significant bits representing `spread` dimensions are aligned with those processor bits that correspond to axes of the network. We present the permutation by means of an example before it is given formally.

We assume that the `spread` dimensions of the given array are sorted so that their relative sizes conform to the relative sizes of the dimensions of the underlying network.

Consider a $(2^4, 2^8, 2^5)$-NN_3 and an array with $p = (2, 7, 5, 3)$ and $s = 3$, $c = 4$, i.e., 2 bits of the first dimension, 7 bits of the second dimension, 5 bits of third dimension, and 3 bits of the fourth dimension mapped onto the processor address. The first three dimensions ($s = 3$) are labeled with `spread`. (The dimensions may have $q_j \neq 0$, but we do not consider the local part of the address in this example.) The bits in the fourth dimension are not relevant to step 4 since `cycle` dimensions do not need neighbor links.

The mapping after step 3 and before considering the topology of the network is:



In general, the bits belonging to a `spread` dimension are not aligned with axes of the given NN_3. To achieve alignment, processor address bits must be permuted.

The first dimension of the array is already aligned with bit position 0 of the processor address. Formally,

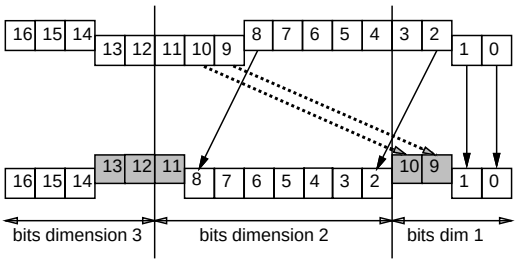
we apply the (identical) permutation

$$\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

The second dimension of the array (shaded above) is not aligned with the bits representing the second axis of the network. This can be achieved by shifting the address left by 2 positions.

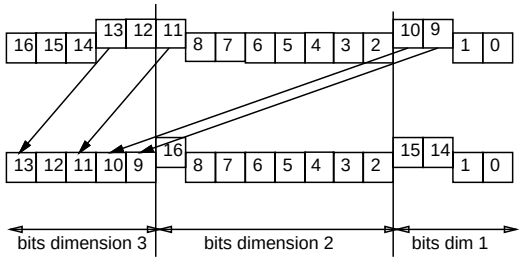
$$\left(\begin{array}{ccccccc|cc} 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 2 & 3 \end{array}\right)$$

The first seven entries in the permutation implement the shift of the bits that belong to the second dimension of the array to the desired position $(4-10)$. The last two entries move those bits that are overwritten by the shift to the released positions.



The final permutation aligns the third dimension (shaded above) with the third axis of the network. Note that due to previous permutations, bits that belong to the third dimension are in various positions of the processor address. Therefore, the results of previous permutations on earlier dimensions must be preserved. Since five bits fit into the third segment of the processor address all five bits $(9-13)$ must be moved. In this example, three target positions must be cleared by moving the occupying bits away.

$$\left(\begin{array}{ccccc|ccc} \pi(9) & \pi(10) & 11 & 12 & 13 & 14 & 15 & 16 \\ = 2 & = 3 & & & & & & \\ & & & & & & & \\ 12 & 13 & 14 & 15 & 16 & \pi(9) & \pi(10) & 11 \\ & & & & & = 2 & = 3 & \end{array}\right)$$



We now present the permutation formally. Let the bits in the processor address be numbered from 0 to $b_p - 1$. Then the following permutation must be applied $1 \leq \mu \leq \min(n, s)$ times, i.e., for each `spread` dimensions the permutation is applied as long as the number of axes is sufficient. Initially, $\pi(i) = i$ holds.

$$\pi'(i) = \begin{cases} j + \sum_{\nu=1}^{\mu-1} \kappa_\nu & \text{if} \quad i = \pi(j + \sum_{\nu=1}^{\mu-1} p_\nu) \\ & \wedge \quad 0 \leq j \leq \min(p_\mu, \kappa_\mu) \\ \pi(i + \min(p_\mu, \kappa_\mu)) & \text{if} \quad \sum_{\nu=1}^{\mu-1} \kappa_\nu \leq i < \sum_{\nu=1}^{\mu-1} p_\nu \\ \pi(i - \min(p_\mu, \kappa_\mu)) & \text{if} \quad \sum_{\nu=1}^{\mu-1} p_\nu \leq i - \min(p_\mu, \kappa_\mu) \\ & \wedge \quad i - \min(p_\mu, \kappa_\mu) < \sum_{\nu=1}^{\mu-1} \kappa_\nu \\ i & \text{otherwise} \end{cases}$$

The first of the above cases maps the appropriate number of the bits belonging to dimension μ to their target positions. Cases 2 and 3 clean target positions by moving the occupying bits away.

Note that again these address computations can be done by simple bit operations. After the permutation is applied, a sequence of bits that originally belong to a single array dimension might run accros several network dimensions in the processor address, especially when mapping low-dimensional arrays on high-dimensional networks, e.g., on a hypercube. In this case performance can be improved by gray-coding of the bit sequence, which is also a fast bit operations.

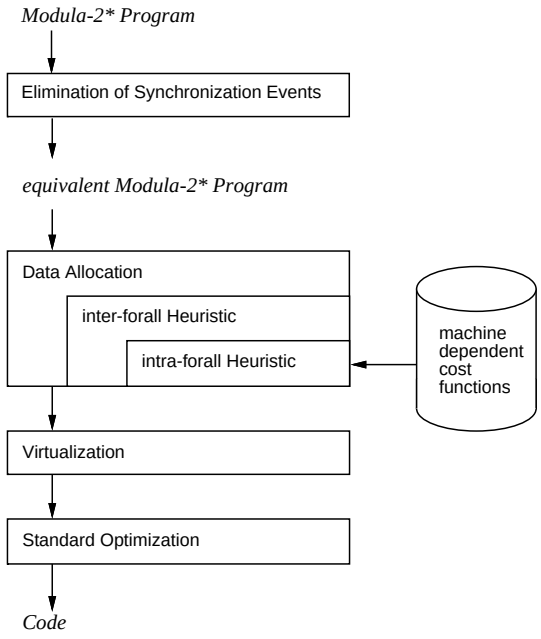
5 Data Allocation in Modula-2*

In Modula-2* (complete definition in [21]) there is no explicit mapping of data elements to processors. Data layouts are derived automatically by the compiler. The mechanism is based on (1) directives in the declaration of the arrays that describe the intended use of individual dimensions of the array on a high level and (2) on the analysis of run-time constants inside of **forall** statements and their use in array subscripts.

In the figure, the general architecture of the optimization phase of our Modula-2* compilers is depicted. The compiler detects redundant synchronization events by data dependence analysis. The elimination process is described in more detail in [15, 6]. This transformation is a prerequisite for an efficient translation for MIMD machines, where synchronization is expensive, and it improves machines utilization on SIMD

machines, since larger code sections can be fused into single virtualization loops.

Data allocation works on a per-procedure basis, i.e., only one procedure at a time is analyzed. To separate machine-independent from machine-dependent parts of the allocation strategy and to limit the size of the problem, we split data allocation hierarchically into two parts.



The inter-forall heuristic, called *Inter* considers the individual **forall** statements as black boxes. Inside a given **forall**, the data distribution is fixed. The inter-forall heuristic decides whether the actual distribution remains unchanged between successive **forall**s, or not. This decision is based on the result of the intra-forall heuristic *(Intra)* which processes the sequence of statements inside a **forall** founded upon known distributions of data elements. The questions are where to compute and to store intermediate results, and where to handle scalar values (front-end, single processor, redundancy, etc.). Whereas *Inter* is machine-independent, because it is based solely on the cost values determined by *Intra*, the latter needs to know about computation costs, the topology of the network, and the communication costs.

Inter is based on Knobe's work [8, 9]: we construct a graph of machine-independent alignment preferences, i.e., we decide which dimensions of two arrays should be aligned to achieve locality of the elements. Alignment is done by constructing super-arrays and by providing functions that map the original arrays onto the super-array. Dimensions of different arrays that are often used together in general end up in the same dimension of the super-array. However, often this graph of alignment preferences contains cycles. Breaking a cycle is equivalent to redistributing array data. Often there are different ways to break cycles, i.e., there are different possible redistributions.

Based on this first phase of alignment, the super-arrays are distributed on the machine by means of the mechanism presented in this paper. The distribution is a prerequisite for realistic estimation of computation and communication costs.

6 Thread Scheduling in Modula-2*

The presented distribution technique can be extended to allocate processors to threads and to determine virtualization. Nested parallelism can be handled as well.

Thread Scheduling

Parallelism in Modula-2* is not limited to simple vector operations. In a **forall** all Modula-2* statements are allowed, e.g., **if**, **while**, procedure calls, and nested **forall**s. Therefore, it is not sufficient to analyze array occurences in a program. Data layout and thread scheduling can be combined by the technique presented above.

A **forall** statement creates a number of threads. Each thread has a run-time constant, to which a unique value of an enumeration type is assigned. These threads must be distributed to the processes. Since a thread is identified by its run-time constant, the array of these run-time constants describes all threads created by the **forall**. Thus, by deriving a data layout for this array, the threads are scheduled.

The analysis of the patterns in which the **forall** constant is used results (1) probably in a linear transformation function reflecting offset and stride of the usage patterns of the run-time constant and (2) probably in an alignment with a particular array dimension used inside of the **forall**, if any.

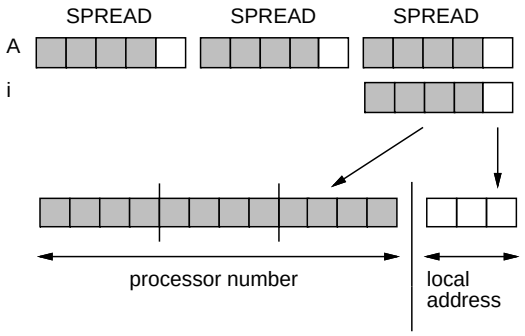
Consider the following example. We assume that `A` is a $32 \cdot 32 \cdot 32$ array, and $N = 32$.

```
FORALL i:[1..N] BEGIN
  A[c,d,i] := 0
END
```

Program analysis gives us that the run-time index `i` which is created by the **forall** should be aligned with the third dimension of array `A`. By applying the data

distribution technique both for the array `A` and the array of run-time constants `i`, we find the following situation.

For the assignment statement of the example only $2^4 = 16$ of the processors are active, which is indicated by the shaded fraction of the address bits of `i`. The white fraction determines the size of the virtualization loops. Since one bit reserved for `i` ends up in the local address, each of the active processors must iterate and perform two assignments. Thus, it is easy to derive from the allocation of `i` to which processor the threads are scheduled and what type of strip mining must be done.



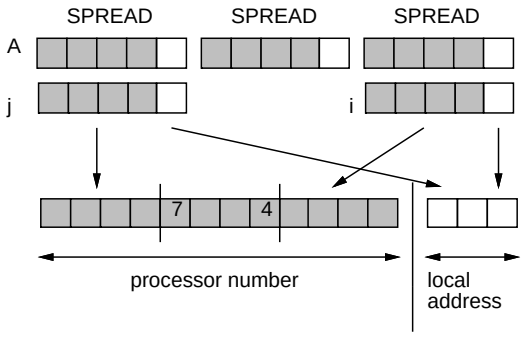
Nested Parallelism

The above concept is neatly extended for both nested and recursive parallelism, as can be seen from a slight extension of the previous example:

```
FORALL i:[1..N] BEGIN
  FORALL j:[1..N] BEGIN
    A[j,d,i] := 0;
    P(...)
  END
END
```

Again, program analysis derives the set of active processors and the iteration space for the virtualization loops.
Consider the call of the procedure `P`. Assume `P` itself spawns additional parallelism. From the layout of the run-time constants it is obvious, which processors must call and virtualize it and which bits of the processor address are free to be used for the additional parallelism. In the example bits 4 to 7 can be used for scheduling of the additional threads.

Because of control statements (**if**, **while**, ...) only a small fraction of the total number of threads may



be active at a procedure call. Thus, not all the bits of the processor address assigned to the initial run-time constants are really required. If the number of active thread needs fewer bits to be coded, rescheduling these threads, i.e., redistributing the context stacks enables a higher degree of parallelism for the called procedure.

7 Conclusion

An important problem facing numerous projects on distributed memory machines is that of distributing array data over the available processors. Although some work has been done on aligning data structures to reach data locality and a sufficient degree of parallelism, by improving the final mapping of the data to the processors, hardware supported nearest neighbor communication can be exploited and address calculations can be simplified. In this paper we have presented an algorithm for mapping an arbitrary, multidimensional array onto an arbitrarily shaped multidimensional nearest neighbor network. This mapping achieves these goals. It has partly been implemented in optimizing compilers for a parallel language, and it supports thread scheduling and the translation of nested parallelism.

References

[1] American National Standards Institute, Inc., Washington, D.C. *ANSI, Programming Language Fortran Extended (Fortran 90). ANSI X3.198-1992*, 1992.

[2] Ingo Barth, Thomas Bräunl, Stefan Engelhardt, and Frank Sembach. Parallaxis (vers. 2) user manual. Technical Report 2/91, Universität Stuttgart, February 1991.

[3] Barbara M. Chapman, Heinz Herbeck, and Hans P. Zima. Automatic support for data distribution. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 51–58, Portland, Oregon, April 28 – May 1, 1991.

[4] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[5] Manish Gupta and Prithviraj Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 43–50, Portland, Oregon, April 28 – May 1, 1991.

[6] Ernst A. Heinz. Automatische Elimination von Synchronisationsbarrieren in synchronen FORALLs. Master's thesis, University of Karlsruhe, Department of Informatics, November 1991.

[7] K. Kennedy and H.P. Zima. Virtual shared memory for distributed-memory machines. In *Proc. of the 4th Conference on Hypercubes, Concurrent Computers and Applications*, volume 1, pages 361–366, Monterey, CA, 1989. ACM Press, New York.

[8] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[9] Kathleen Knobe and Venkataraman Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In Joseph JáJá, editor, *Frontiers '90:The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, University of Maryland, October 8–10, 1990.

[10] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 177–186, March 1990.

[11] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In Joseph JáJá, editor, *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, College Park, University of Maryland, October 8–10, 1990.

[12] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, September 1990.

[13] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[14] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford Science Publications, 1990.

[15] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

[16] Prentice Hall, Englewood Cliffs, New Jersey. *INMOS Limited: Occam Programming Manual*, 1984.

[17] J. Ramanujam and P. Sadayappan. Access based data decomposition for distributed memory machines. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 196–199, Portland, Oregon, April 28 – May 1, 1991.

[18] M. Rosing, R. Schnabel, and R. Weaver. DINO: Summary and example. In *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 472–481, Pasadena, CA, 1988. ACM Press, New York.

[19] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Reference Manual, Version 5.0*, 1988.

[20] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Language Reference Manual*, April 1991.

[21] Walter F. Tichy and Christian G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report No. 4/90, University of Karlsruhe, Department of Informatics, January 1990.

[22] U.S. Government, Ada Joint Program Office. *ANSI/MIL-Std 1815 A, Reference Manual for the Ada Programming Language*, January 1983.